A description of a dialog to enable interaction between interaction tools and 3D objects in collaborative virtual environments

Laurent Aguerreche, Thierry Duval, Bruno Arnaldi

A description of a dialog to enable interaction between interaction tools and 3D objects in collaborative virtual environments

Laurent Aguerreche [1], Thierry Duval [2], Bruno Arnaldi [1]

[1]INSA de Rennes, 20 avenue des buttes de Coësmes, F-35043 Rennes
[2]Université de Rennes 1, Campus de Beaulieu, F-35042 Rennes
CNRS, UMR IRISA, Campus de Beaulieu, F-35042 Rennes
Université Européenne de Bretagne, France

{laurent.aguerreche | thierry.duval | bruno.arnaldi} @ irisa.fr

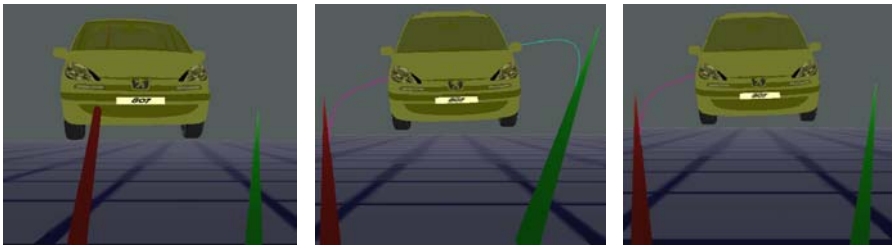

Figure 1: At left, only one ray (the one at left) is controlling car position. At center, both rays are bent (see lines coming from them) because they attempt to modify car position at the same time in opposite directions. At right, the right ray left its control but the left ray continues its action. (Car model courtesy of PSA Peugeot Citroën)

ABSTRACT

Building virtual reality applications is still a difficult and time consuming task. Software developers need a common set of 3D widgets, hardware device abstraction and a set of software components that are easy to write and use. These means are intended to provide collaborative interactions in rich virtual applications, easy use of many input devices, and easy deployment on multi-sites for shared environments.
We propose in this paper a new formalism for 3D interactions in virtual environments to define what an interactive object and an interaction tool are, and how these two kinds of virtual objects can communicate together.
As a consequence, we describe a communication protocol between interaction tools and interactive objects. We then obtain users on different sites that are able to interact in a shared environment with interactive objects that are provided with access levels.
Moreover, this protocol introduces interoperability between VR platforms.
Finally we explain how we implement this protocol with aggregations of reusable small software components to ease development of VR applications.

- **KEYWORDS:** Virtual reality, Collaborative 3D Interactions, CVE, virtual reality platform

1 INTRODUCTION

From a developer point of view, building virtual reality applications is still difficult as opposed to the development of 2D applications as stated by [5, 15]. In fact, building 2D applications can be a matter of an assemblage of a set of well known 2D elements in a graphical application. At the same time, many VR platforms still not propose a common set of 3D widgets but sometimes the connection of low level blocks in an editor (e.g. the Virtools platform).
To ease the development of VR applications, many authors describe abstractions of hardware devices (e.g. [9, 14]). With 2D applications, people generally use a 2D mouse and a keyboard whilst high-level VR applications propose optical markers for 3D tracking, haptic devices, speech recognition, *etc*. Hardware devices need an abstract code layer to ease their deployment, use and replacement by other hardware devices. Also, the application code is built of this abstract layer in order to not depend directly on a very specific hardware device.

This goal has led to the development of many metaphors for interactions (virtual rays [16, 3], virtual hands, Go-Go [16], *etc.*) that are driven by the abstract hardware layer. From a user point of view, it is possible to replace a hardware device, for instance a 3D mouse, with another one, for instance a 3D optical tracker, while still using the same interaction metaphor. Also, interaction tools are equivalent to the pointing or manipulating techniques found with 2D applications.

An interactive object is any object in a virtual environment that users can interact with. An interaction tool [2] is an object in a virtual environment that will send data to an interactive object; a tool will also receive data from an interactive object in order to track its behavior. Finally, an interaction tool may be driven by hardware devices in order to let a user interact with an interactive object, or an interaction tool may be driven by a software component (e.g. a virtual agent with a behavior). As a consequence, a VR application contains interaction tools and virtual interactive objects.

Let us take a simple scenario in order to illustrate our goals. There is a developer, named Tom. Tom builds a virtual environment by connecting small software components that already exist. As a software developer, Tom has sometimes to write new software components. However, his new software components should be easy to reuse. Currently, Tom edits a text file but a graphical editor may replace it later. He sets access properties for objects, their behaviors, their geometries, *etc*.

There are two users. Leponge lives in Paris. He downloads data that Tom produced and stores them on its computer. Bob lives in Laval and downloads also data provided by Tom. Previously, Tom set that the VR application would run on two computers: one for Leponge and another one for Bob. Leponge launches our platform which launches also automatically our platform at Bob's house. Leponge and Bob are sharing the same world. They can carry together an object at the same time depending on the access levels that Tom set for this object. In fact, Tom defined that Leponge has a lower access level than Bob so he cannot move some objects. But, Bob can lower the access level of an interactive object to give it access to Leponge, or he can raise the access level to forbid the control. To interact with objects, Leponge uses sometimes a virtual hand or a virtual ray while Bob uses always a virtual hand. Leponge's metaphors are driven by an optical tracking system. Bob uses a 3D mouse to drive his virtual ray.

In this paper, we propose a communication protocol to describe the messages that a tool and an interactive object will exchange. This protocol provides four advantages:

- For developers:
  - Ease development of metaphors: Tom spends most of his time to put together elements to make metaphors
- For users:
  - Natural interactions: Bob and Leponge can interact together at the same time on objects, there is no need for a mutual exclusion system
  - Multi-sites interactions: Bob and Leponge do not need to be present at a same place to interact in the same world
  - Easy deployment: our protocol aims to provide VR platform interoperability so Bob and Leponge can share the same word and use different platforms. As well, abstraction of hardware devices let Bob and Leponge to use different kind of devices and change them at runtime depending on tasks.

As we stated, many VR applications are written from scratch instead of reusing existing solutions. Usually, they are written around a 3D engine (like OpenInventor, OpenSG, Ogre or JoGL), or a toolkit such as ARToolkit for augmented reality applications. In addition, applications based on VRML mix up in the same graph scene parts to describe hierarchies of geometric models and behaviors/constraint of the interactive elements. We propose to use a set of software components that have to be aggregated together. The dependencies between these elements are described through the connection of these elements in a configuration file. These extensions have been implemented and tested with OpenMASK [13] but can be implemented on other platforms.

Now, we will describe communications between interaction tools and interactive objects, and how they are implemented. The structure of this paper is as follows: In section 2, we present related work, focusing on hardware device abstraction, interaction tools programming and reusability. In section 3, we compare our approach to previous methods. In section 4, we describe our model: interaction tools, interactive objects and how they are linked together. In section 5, we describe the communication protocol with different cases of interaction. In section 6, we propose an implementation of interaction tools and interactive objects to obtain a set a software component to ease their use in many VR applications. Finally, we conclude and then we give some clues leading to future work.

2 RELATED WORK

This section presents related work for interactions with VR platforms. The first part deals with hardware device abstraction which enables the development of many interaction metaphors that we present after. Finally, we also explain that awareness is required to help users or the system to make collaborative interactions possible.

2.1 Hardware device abstraction

The large variety of hardware devices in AR and VR makes them hard to use for software developers. Previous work has undergone to provide a set of basic components to obtain platform-independent code that can speed up the creation of new VR platforms [6].

DWARF [1] is a framework of reusable distributed services described through XML. For instance, the service manager will connect a service providing a captured picture, thanks to a camera, to a system

analyzing the picture for tracking. Services managers running at each system site communicate with CORBA.

MORGAN [14, 6] is a framework for AR/VR that classifies hardware devices within a hierarchy. An hardware device is then derived of a set of classes. For instance, a mouse is a child of the classes *MousePointer*, *Button* and *Wheel*.

Figueroa et *al.* [9] present a software architecture using *filters* connected in a dataflow. A filter that read data from an hardware device encapsulate code to acts as an hardware driver. For example, there might be a 2D mouse filter. Also, other filters can be connected to its outputs to read the data it sends.

OpenTracker [17] is also based on a dataflow mechanism where *filters* are here named *nodes*. A set of connected nodes forms interaction tools as stated by [9]. Each node is made up of one to many inputs whereas it has only one output. Inputs and outputs are both typed and OpenTracker allows a connection from an output to an input only if they are of the same type.

2.2 Interaction metaphors

Interaction tools depend on the abstracted hardware layer and are involved in metaphors commonly found in VR applications. A taxonomy of the main interaction metaphors is provided in [4].

2.2.1 Implementation

Metaphors usually depend on available devices as stated by [5] when no abstract layer is provided for hardware. However, many of them, like hand metaphors (e.g. “Go-Go” [16]) and pointer metaphors (e.g. ray-casting [16, 3]), need to obtain information about the interactive object they want to interact with, for example the object’s position. Figueroa et *al.* [9] use a set of inputs and outputs for data, but they seem to hard-code connections between tools and interactive objects and to limit interactions to the control of the object’s positions.

2.2.2 Instantiation

As a result of hardware device abstraction, many higher-level interaction paradigms have been implemented. With dataflow based platforms, they are composed of a set of connected components (e.g. filters or nodes). A graphical system can be used to connect the objects of a scene. Unit [15] let a user compose its tools via the connection of their properties. InTml [8] relies on a set of interconnected properties and define a XML format, as an X3D extension, to describe 3D interactions. CONTIGRA [7] proposes also an XML-based format, as a X3D extension, based on its own scenegraph. Finally, authoring tools has been developed, which can use for instance a XML-based format to store produced virtual worlds, to hide low-level/technical issues.

2.3 Feedback to the user

From the interactive objects’ side, Smart Objects [12] encapsulates within the object descriptions of its characteristics, properties, behaviors and the scripts with each associated interaction [20]. With such an approach, an interactive object can give useful information to the interaction system in order to provide helpful feedback to the end-user.

Many authors demonstrated that feedback is required to help users. Firstly a user needs to understand the actions she is able to apply to an interactive object: object position update, orientation update, color update, scale transformation, *etc*. Visual metaphors using arrows or cursor indicate that positions can be updated. These visual metaphors can then be used during interaction and be merged with other modalities [21]. Feedback helps a user to understand what her action causes. In a collaborative virtual environment, users must also understand the actions done by other users to help them, to continue theirs actions or to do a simultaneous action [11].

The following section compares our approach with the existing literature about behavior coding, feedback to the user / awareness coding and VR software architecture.

3 OUR APPROACH

We want to enable distributed interactive sessions with multi-users at different sites but sharing the same virtual environment. Each user will interact with interactive objects through interaction tools.

Interaction tools and interactive objects are equivalent to *filter* [9], or *nodes* [17]. We name them *virtual objects*. Our platform, OpenMASK [13] manages low-level communication between them. We propose a communication protocol for these objects in order to:

- Ease development of metaphors: for an interaction tool, it is not needed to develop a specific code for interaction with an interactive object of type “A”, then code for an object of type “B”, *etc*. Dialog between interaction tools and interactive objects is normalized
- Change dynamically access permissions to interactive objects at run-time and/or before in a configuration file
- Provide more dynamicity to interactive objects: a property can be added or removed dynamically. For instance, an interactive object may be static because it does not offer a position attribute. But, this interactive object will become movable if it adds such an attribute after it received a particular event for instance
- Ease deployment of virtual reality platforms: the VR platform does not need to be programmed in a specific language because our protocol defines its own introspection mechanism and does not

need to use technologies such as Java remote method invocation. Our platform is currently implemented in C++. Plus, an introspection mechanism makes the use of a distributed scene-graph which describes properties, not needed unlike [19]; it does not also require complex systems like distributed shared memory models [22] to spread properties

- Provide interoperability between different VR platforms at run-time: a language neutral system is easier to port to many programming languages
- Propose a new communication protocol for communication between interaction tools and interactive objects. In fact this protocol, while more adapted to distributed applications, works also on a single host. This protocol introduces a small overhead to the connection phase since a tool and an interactive object have to become interconnected. When a tool's property has been connected to an interactive object's property, the tool sends data as it would if the connection was hard-coded.

In this paper, we explain also how we implement the protocol communication with virtual objects which are used as empty shells that encapsulate *software extensions*. A software extension is a software component that needs a virtual object as a host. Each of them will be executed sequentially at run-time by its virtual object. Once software extensions have been assembled in an interaction tool, VR application designers can therefore use it instead of a set of many small virtual objects and speed up building of VR applications. For VR application developers, extensions introduce many advantages:

- Improving software engineering: the models proposed in [9, 17] lead to the creation of for-general-use filters/node. For instance, if you need a very specific filter for an hardware device, you will create a filter of type *SpecificFilterForMyDevice* and it will look like any general filter or node while it is not. Software extensions clearly describe pieces of code that are aimed to be used inside a main structure (*i.e.* a virtual object)
- Improving performances at run-time: our platform is multi-sites, in which case we can improve performances if objects are correctly distributed. For this reason, extensions can help us toward this goal because they lead to a unique data structure to distribute, a virtual object, rather than a set of many, filters/nodes. Extensions are automatically moved with a virtual object
- Improving scalability/adding dynamicity: it may be interesting to not add heavy and complex behavior on some virtual object every time to lower CPU or network usage. Therefore, a mechanism, such as our software extensions, introducing a way to add or remove software components at run-time is required. Moreover it must be easy to use. Let us take a very simple filter that will send positions to move an object up and down. To change this behavior, another filter, for instance a filter to move from left to right, will be required. First, it will be need instantiate this filter, if needed, then to disconnect the up-and-down filter, connect the object to the left-to-right filter and eventually remove the useless up-and-down filter. Moreover, creation, destruction, connection and disconnection will have to be managed by a supervisor, so another filter. With software extensions, one object will manage itself its up-and-down extension, after it received an event for instance, with a left-to-right extension.

4 MODEL: TOOLS AND INTERACTIVE OBJECTS

We define an interaction as a bidirectional communication between a tool and an interactive object: a tool sends commands to an interactive object that is responsible to treat the commands and it also receives commands from interactive objects.

4.1 Tools

We assume that an interactive object is only manipulable through a tool. Our assumption leads to a situation where every user who wants to interact with an interactive object will have to use a tool. A user will interact with an object through a tool: a hand, a ray, a pointer, *etc*. This user can be a person or a software component.

A tool modifies properties of an interactive object so it is made to send data to an interactive object that will treat these data. A tool is made up of a set of attributes where each of them has a type and an access level. They define data that a tool will be able to send. For instance, a ray intended to move an object would need to embed a position attribute. The position attribute of the tool contains the position value and sends it to the interactive object when the value is updated.

From the tool's point of view, the communication flow with an interactive object follows three steps: **i**) initialization, **ii**) use, **iii**) release of resources. Those three steps fit with how an interaction happens: **i**) selection of an object, **ii**) manipulation, **iii**) object release. At selection, a tool starts a particular communication with an object designated by a user with a tool (e.g. a ray, a menu, *etc*.). Then the tool requests the possibilities of interaction the interactive object offers and presents them to the user. The user can now start manipulation on some object properties: we call this state the *manipulation state*. The tool is now sending values to the interactive object (e.g. a new position, a color). If many tools are sending values to an interactive object, we say that those tools are interacting cooperatively and so the interactive object

will have to deal with those concurrent inputs, we will explain this case in the following section.

4.2 Interactive objects

An interactive object must give some knowledge to tools about control it offers. Our model assumes that each interactive object embeds a set of attributes which define the data the object can receive. An interactive object can be interrogated to give its interaction capabilities.

When a tool knows the capabilities an interactive object offers, it can take control of some of the attributes to modify the data they contain. As a consequence, any tool can modify any properties of an object. We define a control interaction access policy for the tools in order to regulate which of them can interact. An interactive object arbitrates actions from the tools: it can accept or refuse any action from them. Also a tool does not have any way to directly stop interaction of a concurrent tool. Moreover, a control can be interrupted. This access policy is associated with attributes of an interactive object.

Each attribute has a type, a shareable flag, an informative entry and an access level. The type and the informative entry are used to match a tool attribute with an attribute of an interactive object when a tool tries to take control of an interactive object attribute. For example, we need to make a tool aware that the object it wants to control position has a position attribute, and we also need to connect the tool to the position attribute.

A shareable attribute must be able to deal with concurrent actions. In this way an attribute is preceded by a *converter*. A converter aims to handle concurrent actions and convert them into a value that the associated attribute will use as a new value to contain. A converter may compute the mean value of a set of positions. We do not propose any way to handle contradictory actions like a Boolean set to *true* by a tool while another tool sets it to *false*. Contradictory actions will usually lead to the last action received being the result stored into the current attribute.

Finally, some tools can be manipulated as interactive objects to move them or change their color for instance, so those tools are both tools and interactive objects at the same time.

4.3 Relations between tools and interactive objects

Figure 2 gives the general picture of how two tools control the position of an interactive object. An *interactor module* is embedded in a tool and has to compute a new position to send to the interactive object. The interactor module must also listen for events coming from the controlled interactive object to track its activity: if a new tool is connecting to the interactive object for instance. Finally, the interactor module implements the tool part of our communication system. An interactive object embeds an *interactive module* which implements the interactive object part of the communication system. The *fusion converter* presents an attribute to let the two tools modify the object position. For positions, it can be limited to the computation of average values positions from *n* values that it received.

The interactor module and the interactive module are both a piece of software that can be added or removed dynamically to any object to turn it into an interactor or an interactive object. We will see that they contain *software extensions*.

5 How to make interaction tools and interactive objects communicate?

This section explains how one interactive object can be manipulated by many tools, then how a tool can manipulate one or many interactive objects.

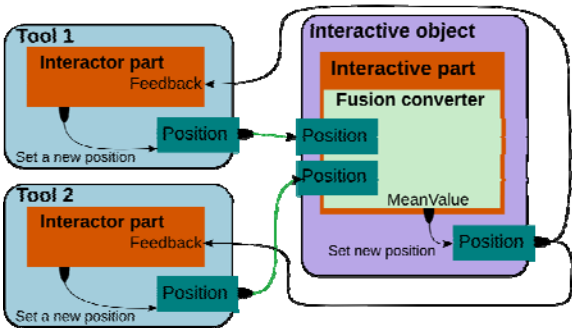

Figure 2: Global organization of two tools controlling position of an interactive object.

5.1 One interactive object with many tools

We consider the two ways an interactive object can be manipulated: first only one tool is manipulating the object; second at least two tools are manipulating the (same) interactive object at the same time.

5.1.1 Simple control

The sequence we describe in this section is illustrated with the figure 3 and shows one tool that will control all the attributes found as accessible in an interactive object. First, the tool opens a session[1] with the interactive object previously selected by the interactor, which can be a human person or a software component.

[1] This step is required for technical reasons: for instance the connection might need some initializations of internal data structures.

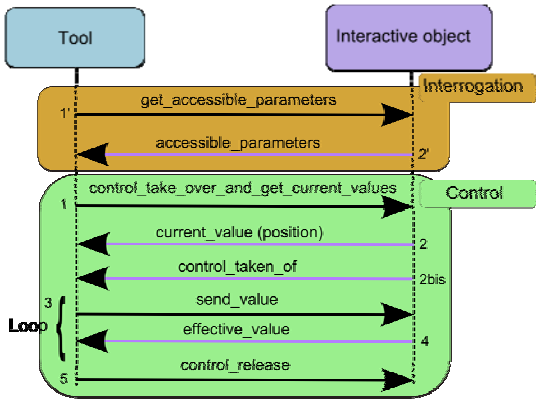

Figure 3: Communication sequence between a tool and an interactive object.

A person may select an object via ray-casting for instance. Then the tool needs to interrogate the interactive object to obtain the object attributes it would be able to manipulate:

- It sends a *get_accessible_parameters* command to the interactive object
- The interactive object answers with an *accessible_parameters* message which is also made up of a list of IDs of accessible parameters of this object for this tool.

This interrogation system does not rely on an introspection mechanism given by a programming language such as Java, but it defines a set of types given by a configuration file for the virtual environment and associates some meta-data to describe the attributes. This way we define a programming language agnostic protocol.

Now the tool can try to take control of some attributes and later to send new values to the interactive object:

- The tool attempts to take control of the accessible parameters. For instance, if it tries to take control of a Position attribute, it will send a *control_take_over_and_get_current_values* message with an ID for the Position
- The interactive object uses a *current_value* message to send back a value for the position; this message is composed of an ID for the position attribute and the position value. When all the values the tools were interested in have been sent then the interactive object sends a *control_taken_of* message, which is composed of the IDs that the tool is controlling. This message acts also as an end flag. Now depending on the values the tool received, it is able to initialize itself for further computations. With a position, a tool can compute an offset between itself and an interactive object to simulate a fixed slider constraint when it moves the object
- The tool can now propose new values to the interactive object with a *send_value* message. The tool will send as many messages as they are values to send
- The interactive object periodically informs a controlling tool of the values of the attributes the tool is controlling with an *effective_value* message. Each message contains the value of a controlled attribute
- The tool can release control of an attribute it is controlling with a *control_release* message. This message is sent with IDs of the attributes to release control of.

5.1.2 Shared control

A *shared control* stands for a control where at least two tools are manipulating the same interactive object. At this time each tool needs: **i**) to know the other tools that are manipulating the same object, **ii**) to track the object's evolutions to inform its users (human person or software component) about incoming/outcoming of tools, **iii***)* or to adapt its internal computations. A human user adapts her actions depending on the actions that other tools are doing: for example, in order to help another user, someone needs to become coordinated with the user or the actions she made, *etc*.

Let us consider a tool named *T*1 which is interacting with an interactive object named *O*. If another tool, named *T*2 takes control of some attributes of *O* then *T*1 will receive a *control_taken_by* message made up of the ID of *T*2 and also the IDs of the controlled attributes. When *T*2 releases the control of some attributes, *T*1 will receive a *control_released_by* message made up of the same kind of attributes a *control_taken_by* message uses.

The figure 1 illustrates how two rays manipulated by two human persons can move a car together. A ray becomes bent when the position it tries to apply is constrained by, for instance, the position given by another tool. This metaphor is described in [18].

The use of the *control_take_over_and_get_current_values* message is not decomposed into two commands to avoid a mutual exclusion. Concurrency is supported by the interactive object itself to let many people interact together. Moreover, if we first ask for control and latter for values, we will introduce an important delay between object picking and the beginning of object manipulation. This is a main issue when a user wants to pick a moving interactive object.

5.2 One tool with many interactive objects

We consider that a tool can manipulate an object if it has an interactive extension. We will explain how one tool interacts with one interactive object. Then, we will explain what a complex interactive object is and how one tool interacts with.

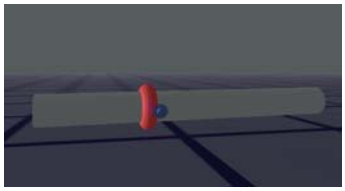

Figure 4: This 3D widget is a slider which is made up of a ring sliding along a bar. Here the ring is selected and we can see a small pointer in front of the ring that a user can manipulate to move the ring.

5.2.1 Control of a single simple interactive object

This case is the one presented in section 5.1.1. Also, a tool takes control of the set of attributes it asked for. After the manipulation of the position of the interactive object, the tool releases its control.

5.2.2 Control of a single complex interactive object

We qualify an object as *complex* when it has a "big" number of attributes to interact with. Such an object requires a particular approach for interaction.

Some virtual objects would require a composition of interactive objects: a "virtual" car may be made up of doors, wheels, an engine, a chassis, *etc*. where each of these pieces embeds an interactive extension. Some constraints would be added to interactive objects thanks to software extensions in order to keep doors attached to the car; maintain seats within the car, *etc*. so an interactor can move the entire car and interact with some of its parts without disassembling everything. The interactive feature is described here through a *local* approach.

Let us consider a flexible hose made up of many rings so it may be hard to add local constraints between them and compute positions. A *global* approach may be needed: only one (or at least a small number) interactive object is employed to implement a finite element method. When an interactor wants to apply some actions on the flexible hose, it has to go through the unique interactive object which computes all the properties of each ring. This feature needs to be able to decompose a flexible hole into a set of rings selectable by the user. Associations between rings and attributes of the hose are described in a configuration file.

A user may select an object part by different ways: a menu, voice, *etc*. As an example, we assume that a user selects an object by ray-casting. A 3D mesh is decomposed into a set of 3D submeshes so a user will pick one of them. Each 3D submesh is associated with a given ID thanks to the configuration file describing the virtual environment. Now, when a user picks a 3D submesh it obtains an ID that it sends through a *get_accessible_parameters* message. The interactive object receives an ID of one of its part, and then it answers with the associated attributes. This message behaves like a filter for the interactive object attributes.

As a simple example, figure 4 shows a 3D widget made up of two parts. At screen, there are two 3D meshes (a ring and a slide) but only one interactive object allows interaction and it keeps these two objects together. If the interactor selects the ring then it will move it along the slider, if it selects the rest of the slider it will move all the slider including the ring.

5.2.3 Control of a set of interactive objects

With a set of interactive objects, a tool does not directly interact with interactive objects but through *proxy-tools* (see figure 5). A proxy-tool is an intermediate tool between the "real" tool, that the user manipulates, and the interactive object. Note that to simplify, we did not decompose tools in a main tool + proxy-tools in section 5. The use of a proxy-tool aims to add more flexibility to how a tool is programmed: the tool does not have to contain code to communicate directly with one interactive object or many. A proxy-tool embeds the code to manipulate an object: a *control extension* and a *behavior extension*. Also it becomes easier to manipulate many interactive objects because the tool only has to instantiate many proxy-tools and then follows their evolution.

Moreover a proxy-tool can have its own extensions that are different with the tool. Those extensions can add a specific behavior to each proxy-tool. In addition, proxy-tools can be spread on the network to reduce latencies and network traffic during a distributed session.

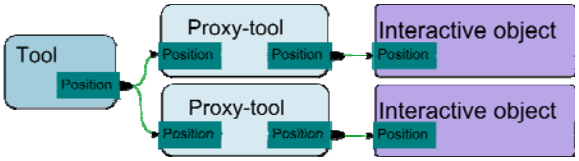

Figure 5: A tool uses proxy-tools to manipulate some interactive objects. The main tool sends its position then each proxy-tool computes a position for an interactive object using the main tool's position.

5.3 Access to interactive object properties

Platforms for virtual reality may be used to simulate real-life interactions where a trainer explains industrial maintenance procedures to technicians, a boss accesses to confidential data, *etc*. Prior work has been done in [10], for instance, but its policies seem quite limited.

Each attribute of an interactive object is provided with an access level. When a tool tries to manipulate an attribute, it first needs to have an attribute that matches the expected interactive object attribute: same type, corresponding meta-data and a sufficient access level. This policy leads to the fact that a tool can fail to take control of some attributes, or it can be "kicked out" by another one. When a tool is interacting with an interactive object, another one can come and raise access level to a higher value, thus the former tool lost its control and receives a *control_ended* message which is made up of IDs of the released attributes. In order to

implement this policy, we propose to manage *interactive properties* that can be associated with interactive objects themselves or interactive object attributes. Let us describe elements of such a property:

- A name, a type (e.g. Integer, Position, Orientation, *etc*.), a meaning (i.e. a kind of comment), an association with a feature of the interactive object (e.g. interactive object position) and the number of tools that can interact with it at the same time
- A priority level in order to determine the lowest priority needed by an interaction tool to be allowed to interact with an object. This priority can be set globally for all tools, or for each existing group of tools, or for an object. The priority can be set globally for an object or specifically for each of its attributes
- An access policy for an interactive object attribute or a group of interactive objects. It determines how the interactions can be shared between many tools. The different policies allow either many interaction tools to share interactions with an interactive object, or only one tool to interact with the object. Currently, we propose three policies:
  - "Unfreezable" policy: a tool interacting with such an attribute will not be able to forbid another authorized tool to stop or join the current interaction. Motivation: a tool will not be "kicked out" by another one
  - "Freezable at any level" policy: a tool interacting with such an attribute will be allowed to determine, for each existing group of tools, the minimum required level needed for a tool in order to be able to stop or join the current interaction. Motivation: a tool will give access to only a set of tools
  - "Freezable at tool level" policy: nearly the same policy than "Freezable at any level" but the level cannot be superior to the priority of the tool that initiated the interaction session. Motivation: a tool will lower the access level to allow new tools. For instance, we imagine a powerful user that give access to some persons by lowering access level
- A description of how the feedback used to make the user aware of its interaction (with an attribute, an object or a group of objects), have to be triggered: before, at the beginning, during or after the selection and/or the manipulation of an interactive object or attribute.

6 PROTOCOL IMPLEMENTATION

This section describes the different software extensions that interaction tools and interactive objects use to add their features and implement our communication protocol.

6.1 Specialization with aggregation

While we usually specialize the behavior of a class (in C++ for instance) by subclassing it, we use instead an approach based on a set of components. Therefore we start from a class which is like an empty shell and we add it a behavior (or we refine its behavior), by aggregation of software components, which are named *software extensions*. This approach enables a fine-grained programming since to obtain a completely new object it just requires to change the set of used extensions. Furthermore it is also possible to refine the behavior of an extension by subclassing it.

This approach is not new in software design but we introduce here a scheme for tool and interactive object use.

6.2 Parts of a tool

We see a tool as an aggregation of, at least, four *software extensions*: *selection*, *manipulation*, *control* and *behavior* (see figure 6).

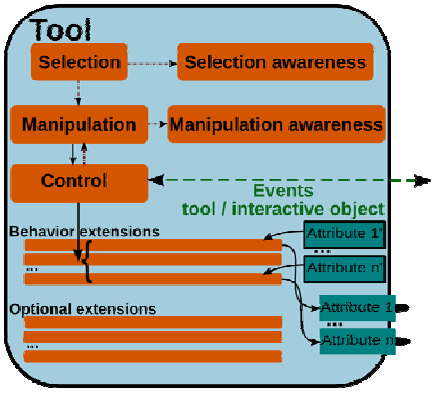

Figure 6: A tool with its attributes and its software extensions.

6.2.1 Selection

This software extension stores IDs of the interactive objects that an interactor selected. If the interactor is a human person then she may use a ray-casting method to designate a set of objects, or she may use a menu, or her voice, *etc*. Since the goal of the *selection extension* is to store IDs of the selected objects, many approaches can be implemented to provide a selection method to a human user. In addition this approach is generic since any virtual agent, driven by a human person or a software component, can select objects.

6.2.2 Manipulation

This software extension listens for incoming events and then orders the *control extension* to do something. The *manipulation extension* also aims to implement how the tool reacts when it receives information from the *control extension*. When the user wants to interact with an interactive object, this software extension will display, for instance, a menu to ask the user which actions she wants to apply. This software extension is also responsible for awareness message routing: is it

worth to inform the user about a particular event while she is interacting for instance?

A human person may begin an interaction if she presses a button of a hardware device, this would send an event that this software extension would interpret. This software extension attempts to interact with previously selected interactive objects, so this software extension uses the interface given by the *selection extension*; here we employ *listeners* with the *selection extension*.

A finite state machine is created every time the corresponding tool tries to take control of an interactive object. In fact, this extension manipulates one finite state machine per interactive object that the tool manipulates, and runs them at the same time and independently. States of a finite state machine are aimed to send commands (*get_accessible_parameters*, *control_take_over_and_get_current_values*, *etc*.) whilst its transitions between states are fired when the tool receives expected replies from an interactive object.

6.2.3 Control

This software extension implements the tool part of the communication protocol so it has four capabilities:

- Open/close a session with an interactive object
- Interrogate an interactive object about its accessible attributes
- Taking/release control of some interactive object attributes
- Track control evolution of other tools: a tool takes control, or release control, of some attributes that the current tool is manipulating.

6.2.4 Behavior

This software extension contains code to compute values to apply to an interactive object. A tool is composed of many *behavior extensions* where each is tied to a tool attribute. When a tool attribute is associated to a manipulated interactive object attribute then its behavior extension is initiated and will run until this attribute stop to send values. A behavior extension is made up of three methods: *initComputes* that is called for initialization, for instance to compute an offset between the tool and the interactive object to move; *compute* that is called each time the attribute tool has to produce a new value to send to the manipulated interactive object; *endComputes* that is called when the attribute has lost its control.

6.2.5 Awareness

We also associate to the *selection extension* a *selection awareness extension* that aims to implement the code to aware a user (a human person or a software entity) that an object has been selected or unselected. Consequently this software extension is not tied to a particular tool: if this software extension simply asks the 3D engine to highlight a selected object then it is possible to use this software extension with different tools.

Similarly we define a *manipulation awareness extension* which will listen the *manipulation extension* for new interactive object attributes being controlled or released. This software extension may be used to highlight a 3D mesh associated to a currently manipulated interactive object for instance.

6.3 Parts of an interactive object

An interactive object is mainly designed to communicate with a tool and thus shares common aspects.

6.3.1 Symmetries with tool software extensions

An interactive object is also composed of an aggregation of software components, which are the *software extensions*. This object requires at least the *interactive extension* which implements the interactive object part of the communication protocol. The interactive extension has to reply to interrogations about accessible attributes, control taking or control release.

Similarly to the awareness extensions of a tool, it is possible to write software extensions that will listen for what happens to the interactive object. This software extension currently exposes an interface with *newInteractor(newToolID)* being called when a new tool takes control of at least one attribute, *leftInteractor(leftToolID)* being called when a tool fully releases its control on an interactive object. This interface is currently quite rough but would be extended in the future to become more informative about the attributes controlled, released, *etc*.

6.3.2 Combination of values sent by tools

The interactive object has to interpret the commands coming from tools and then store the result in one of its attribute.

In order to combine values coming from many tools, each interactive object attribute is provided with a *converter* which is a piece of software that takes a set of values and returns a combination. A usual converter may computes the mean value of many positions given by many tools, or it may compute a sum of values, add noise to the values, or apply some transformations.

7 CONCLUSION

Our motivation is to help software developers to produce rich virtual reality applications, with hardware input abstraction device, software component reuse, easy modification of instantiations of virtual applications. As a result, we propose a new formalism for 3D interactions in virtual environments. This formalism defines what a virtual interactive object and an interaction tool are, and how these two kinds of objects can communicate together. We show how this communication system works for simple interactions:

one tool interacting with one interactive object; and more complex collaborative interactions: two tools interacting simultaneously with the same interactive object, making the users aware of this closely coupled collaboration.

For software reuse, we have developed many software components, which are called *software extensions*, dedicated to selection and manipulation. The *manipulation extension* also uses a *control extension* which implements the communication protocol we described. Interactive objects embed an *interactive extension*. Thereby tools and interactive objects are easier to implement and describe.

The formalism we introduced is a first step toward a description language to describe the interactive and collaborative properties of virtual objects.

8 FUTURE WORK

The communication system and the description of what a tool and what an interactive object are, is a step towards a more sophisticated language to describe interactions in virtual reality applications. The description we propose is based on the use of many software components which are assembled.

Furthermore we also think that the union of these format extensions and the description of the communication dialog for tools and interactive objects can enable a better interoperability between different virtual reality platforms.

9 ACKNOWLEDGMENTS

This work was supported in part by the grant 06 TLOG 031 of ANR, in the context of the RNTL-Part@ge project.

REFERENCES

[1] Design of a component-based augmented reality framework. In ISAR'01: Proceedings of the IEEE and ACM International Symposium on *Augmented Reality (ISAR'01)*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[2] D. A. Bowman and L. F. Hodges. User interface constraints for immersive virtual environment applications. Technical report, Graphics,Visualization, and Usability Center GIT-GVU-95-26, 1995.

[3] D. A. Bowman, D. B. Johnson, and L. F. Hodges. Testbed evaluation of virtual environment interaction techniques. *Presence: Teleoper. Virtual Environ.*, 10(1):75–95, 2001.

[4] D. A. Bowman, E. Kruijff, J. Joseph J. LaViola, and I. Poupyrev. *3D User Interfaces: Theory and Practice*. Addison-Wesley/Pearson Education, 2005.

[5] W. Broil, J. Herling, and L. Blum. Interactive bits: Prototyping of mixed reality applications and interaction techniques through visual programming. *3D User Interfaces, 2008 IEEE Symposium on*, pages 109–115, March 2008.

[6] W. Broll, I. Lindt, J. Ohlenburg, I. Herbst, M. Wittkämper, and T. Novotny. An infrastructure for realizing custom-tailored augmented reality user interfaces. *IEEE Transactions on visualization and Computer Graphics*, 11(6):722–733, 2005.

[7] R. Dachselt, M. Hinz, and K. Meißner. Contigra: An XML-based architecture for component-oriented 3D applications. In *Web3D '02: Proceedings of the seventh international conference on 3D Web technology*, pages 155–163, New York, NY, USA, 2002. ACM.

[8] P. Figueroa, M. Green, and H. J. Hoover. InTml: A description language for VR applications. In *Web3D '02: Proceedings of the seventh international conference on 3D Web technology*, pages 53–58, New York, NY, USA, 2002. ACM.

[9] P. Figueroa, M. Green, and B. Watson. A framework for 3D interaction techniques. In *CAD/Graphics'2001*, volume 8, pages 22–24. International Academic Publishers, 2001.

[10] C. Greenhalgh, J. Purbrick, and D. Snowdon. Inside MASSIVE-3: flexible support for data consistency and world structuring. In *CVE '00: Proceedings of the third international conference on Collaborative virtual environments*, pages 119–127, New York, NY, USA, 2000. ACM.

[11] C. Gutwin and S. Greenberg. The effects of workspace awareness support on the usability of real-time distributed groupware. *ACM Trans. Comput.-Hum. Interact.*, 6(3):243–281, 1999.

[12] M. Kallmann and D. Thalmann. Modeling objects for interaction tasks. In *Computer Animation and Simulation '98*, pages 73–86, 1998.

[13] X. Larrodé, B. Chanclou, L. Aguerreche, and B. Arnaldi. OpenMASK: an open-source platform for virtual reality. In *IEEE VR workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, Reno, NV, USA, 2008.

[14] J. Ohlenburg, I. Herbst, I. Lindt, T. Fröhlich, and W. Broll. The MORGAN framework: enabling dynamic multi-user AR and VR projects. In *VRST '04: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 166–169, New York, NY, USA, 2004. ACM.

[15] A. Olwal and S. Feiner. Unit: Modular development of distributed interaction techniques for highly interactive user interfaces. In *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 131–138, New York, NY, USA, 2004. ACM.

[16] I. Poupyrev,M. Billinghurst, S.Weghorst, and T. Ichikawa. The Go-Go interaction technique: Non-linear mapping for direct manipulation in VR. In *Proceedings of the 9th annual ACM symposium UIST*, pages 79–80. ACM Press, 1996.

[17] G. Reitmayr and D. Schmalstieg. An open software architecture for virtual reality interaction. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 47–54, New York, NY, USA, 2001. ACM.

[18] K. Riege, T. Holtkamper, G. Wesche, and B. Frohlich. The bent pick ray: An extended pointing technique for multi-user interaction. In *3DUI '06: Proceedings of the 3D User Interfaces (3DUI'06)*, pages 62–65. IEEE Computer Society, 2006.

[19] D. Schmalstieg. Distributed applications for collaborative augmented reality. In *Proceedings of IEEE Virtual Reality 2002*, pages 59–66, 2002.

[20] P. Sequeira, M. Vala, and A. Paiva. What can I do with this?: Finding possible interactions between characters and objects. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–7. ACM, 2007.

[21] J. Sreng, A. Lécuyer, C. Mégard, and C. Andriot. Using visual cues of contact to improve interactive manipulation of virtual objects in industrial assembly/maintenance simulations. *IEEE TVCG*, 12(5):1013–1020, 2006.

[22] H. Tramberend. Avocado: A distributed virtual reality framework. In *VR '99: Proceedings of the IEEE Virtual Reality*, page 14, Washington, DC, USA, 1999. IEEE Computer Society.